2019-2020

Architecture Orientée Services (PARTIE A)

JAVA

LABIDI Mohamed
ISET SOUSSE | GLDRA1

Introduction

Quand vous développez une application en Java (ou autre) en entreprise, il est rare que vous commenciez dans un système vierge à partir d'une page blanche et encore plus que vous soyez le seul à travailler dessus.

Vous arrivez donc en général dans un système composé de dizaines voire de centaines d'applications, qui s'envoient des messages, communiquent et qui peuvent se trouver dans des serveurs locaux, dans des serveurs distants ou dans le cloud. Elles sont aussi écrites dans divers langages : Java, C#, C++, Python ou pire encore COBOL et autres hiéroglyphes.

Se posent alors plusieurs problèmes :

- Dans le SI (système d'information) d'une entreprise, chaque application peut avoir son propre protocole de communication, ses propres formats de messages, etc. Se pose alors la question de comment créer votre application de façon qu'elle communique parfaitement avec les autres ? Ou encore mieux, comment concevoir toutes les applications du SI dès le départ pour qu'elles communiquent toutes via un protocole et un format standard et compris de tous ?
- Comment éviter de réinventer la roue : savoir si certaines fonctionnalités existent déjà dans d'autres applications ou chercher la possibilité de **réutiliser** certains composant afin d'éviter la redondance.
- Comment savoir à quelle URL ou IP se situe telle ou telle application avec laquelle vous voulez communiquer ? Pire encore, que se passe-t-il si cette application change d'URL ?

Ce sont des exemples parmi d'autres de problèmes auxquels les développeurs étaient confrontés avant le développement de l'architecture orientée services (SOA).

Vous allez voir dans ce cours comment les entreprises organisent leurs SI afin que toutes les applications communiquent très facilement via un protocole unique appelé SOAP, se mettent à jour sans interruption de service et redeviennent à taille humaine grâce à un découpage de l'ensemble en petites applications appelées services.

Vous allez ensuite apprendre à découper une application en services, puis vous apprendrez à créer et tester ces services dans le respect des principes de la SOA.

Le but ici n'est pas que vous soyez capables de créer une architecture SOA à vous seuls, mais plutôt que vous sachiez créer des services qui s'intègrent dans une SOA existante. C'est ce qui vous sera la plupart du temps demandé quand vous intégrerez une entreprise dont le SI est basé sur cette architecture.

Objectifs pédagogiques :

- Concevoir une application web avec une approche par composants
- Créer un web service SOAP
- Isoler et déployer une application dans des conteneurs grâce à Docker
- Sélectionner les langages de programmation adaptés pour le développement de l'application
- Interagir avec des composants externes

PARTIE 1 : Pourquoi mettre en place une SOA

Pour comprendre et assimiler les notions d'architecture orientée services (SOA), nous allons suivre tout au long de cette partie une petite entreprise fictive du nom de **ISETTECH** qui fabrique et vend des jouets électroniques.

Créée au début des années 1990, cette entreprise a connu une croissance continue jusqu'à nos jours. Son catalogue a évolué au fil des années et ses clients se sont diversifiés.

L'entreprise a donc été amenée à faire évoluer son SI afin de s'adapter à cette croissance et aux nouveaux canaux de vente comme Internet.

Comment faisions-nous avant ?

Jusqu'à la fin des années 1990, les entreprises exploitaient dans leurs systèmes d'informations (SI) des mainframes qui s'occupaient de toutes les opérations.

Ainsi, notre entreprise **ISETTECH** avait commencé par mettre en place un mainframe, une sorte de grand ordinateur central très puissant (pour l'époque) et qui pouvait traiter un très grand nombre de requêtes, voire faire tourner plusieurs sessions d'un système d'exploitation.

Les développeurs déployaient alors leurs programmes dans cet ordinateur unique. Comme le système était unifié et que tous les programmeurs travaillaient plus ou moins dans le même environnement et avec les mêmes technologies, tout fonctionnait dans une joyeuse harmonie !

Mainframe IBM

Revenons à notre entreprise. Au début des années 2010, **ISETTECH** a rencontré un problème : les procédés dans l'entreprise s'étant informatisés de plus en plus, il a fallu intégrer au SI de plus en plus de logiciels externes à forte valeur ajoutée.

Par exemple, les dirigeants souhaitaient :

- Intégrer un ERP de gestion du service après-vente (SAV),
- Mettre en place un e-commerce grâce à un CMS pour vendre les produits en ligne et profiter de l'essor de ce secteur,
- Automatiser la gestion des commandes des clients professionnels (les grossistes) afin de rester dans la course face à une concurrence de plus en plus performante, rapide et informatisée.

Chacun de ces systèmes à ajouter avait besoin d'un environnement adapté : système d'exploitation, capacité de stockage et de calcul, configuration spécifique (ports, pare-feu, etc.).

La solution était d'abandonner le bon vieux mainframe, devenu très limitant, et de le remplacer par plusieurs ordinateurs (serveurs) munis d'environnements adaptés à chaque système.

Ainsi **ISETTECH** se retrouvait alors à exploiter :

- Un serveur pour gérer la boutique en ligne,
- Un serveur pour l'ERP du SAV,
- Un serveur pour les grossistes et la gestion des commandes,
- Un serveur pour un ERP de comptabilité.

ISETTECH se trouvait ainsi confronté à un gros problème : **comment faire communiquer tous ces différents systèmes ?**

La réponse semblait simple : faire appel à chaque système dans le format qu'il comprend.

Ainsi, quand l'ERP, écrit en COBOL, doit appeler la boutique en ligne, écrite en PHP, pour obtenir la dernière commande d'un client, la requête doit être envoyée dans un format *X* compatible PHP. L'ERP reçoit alors de la boutique une réponse dans un format *Y* qu'il doit transformer en quelque chose de compatible avec COBOL avant de l'exploiter.

Ces transformations sont réalisées par des adaptateurs. Chaque logiciel dans le SI doit avoir un adaptateur pour communiquer avec un autre.

Finalement, à mesure que le système grandissait, on se retrouvait avec les problèmes suivants :

- Le système devenait trop complexe : chaque logiciel devait gérer des dizaines d'adaptateurs pour communiquer avec des dizaines d'autres logiciels, qui eux-mêmes disposaient d'autres adaptateurs.
- Flexibilité limitée : changer la moindre fonctionnalité dans un logiciel du SI impliquait des changements en cascade dans tout le SI : les adaptateurs à mettre à jour, un redéploiement de l'ensemble, etc.
- Scalabilité difficile.

- Coût de maintenance trop élevé dû à la complexité du système et à l'interdépendance forte entre les composants du SI.

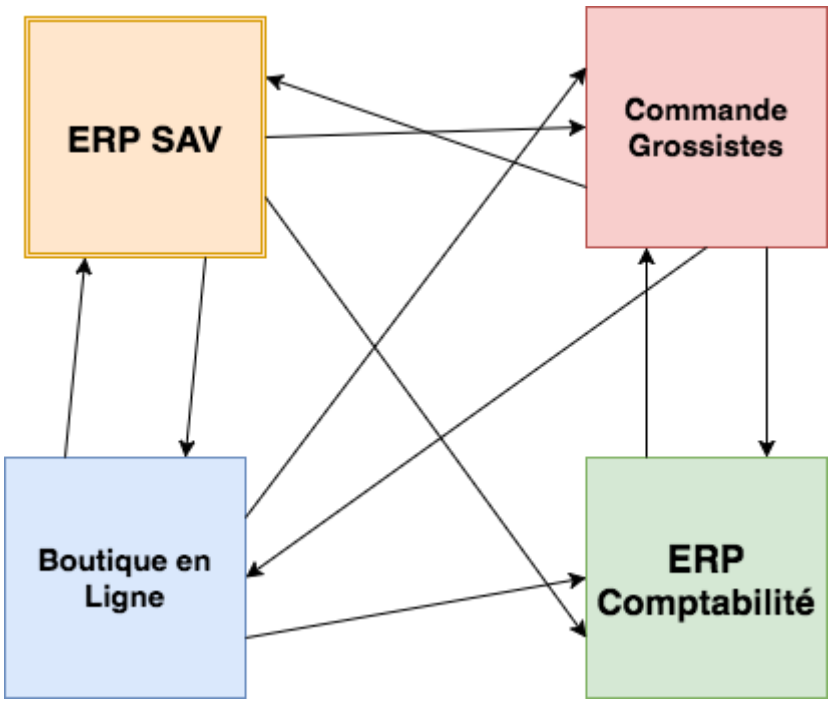


Diagramme simplifié des systèmes avant la SOA

La solution grâce à la SOA

Avec l'arrivée du XML, l'architecture orientée services a fait son apparition.

L'idée est simple : organiser les SI de façon qu'ils soient composés de briques indépendantes appelées **services**. Chaque service a un nombre de fonctionnalités cohérentes et est indépendant des autres services.

Ces services vont communiquer entre eux grâce à un protocole standard, connu et compris de tous. Le protocole qui s'est largement imposé est le SOAP basé sur le XML. On y reviendra en détail.

Et concrètement ?

Pour bien comprendre le concept, analysons une implémentation de la SOA sous forme d'un web service.

Il est important de distinguer l'architecture orientée objet, qui est un ensemble de théories et de bonnes pratiques, des web services qui sont eux une implémentation ou une application des concepts théorisés dans une SOA.
Imaginez que vous développez un web service qui doit permettre de gérer les clients d'une banque.

Ce web service va avoir les fonctionnalités suivantes :

- Récupérer un client grâce à son ID.
- Mettre à jour les informations d'un client.

- Récupérer ses mouvements de compte.

Vous développez le service en Java qui ira interagir avec les bases de données de la banque pour accomplir les fonctions citées plus haut.

Une fois mis en production, votre service va être appelé par un autre service qui s'occupe de l'application mobile. Il a besoin de faire appel aux informations que propose votre service pour afficher une page de compte à l'utilisateur final.

1er problème

Comment indiquer au service mobile où trouver votre service, les fonctionnalités qu'il propose et comment les appeler ?

La réponse est simple : il faut établir un "contrat" entre les services qui explique clairement comment fonctionne chaque service.

Ce contrat a le plus souvent le format d'un document XML appelé WSDL. Sa structure est standard et connue de tous les services.

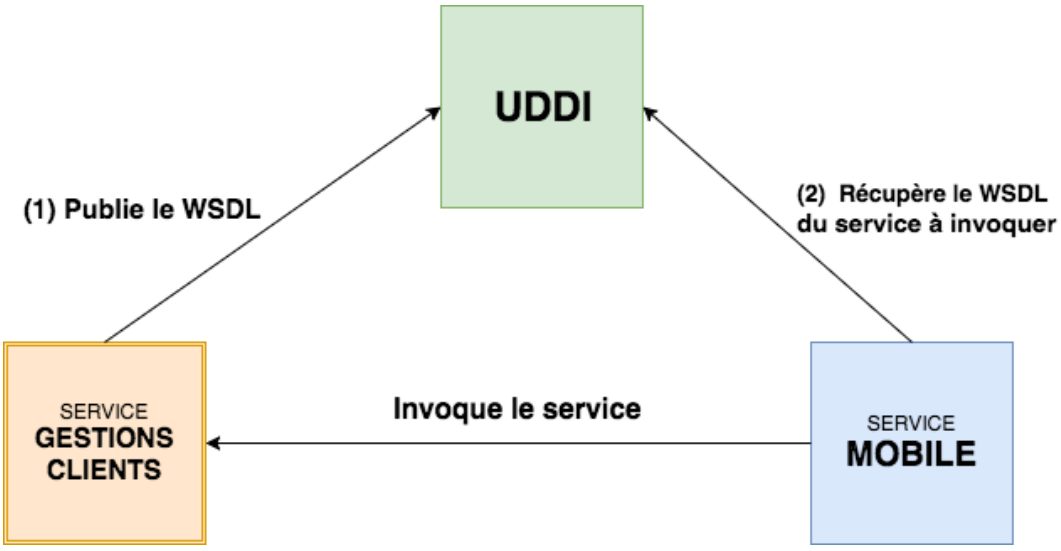
Vous rédigerez donc un document WSDL dans lequel vous décrirez l'URL où on peut trouver votre service, quelles fonctions vous proposez et quels paramètres sont requis pour chaque fonction. Le service mobile n'a plus qu'à consulter ce document pour pouvoir faire appel à votre service en toute fluidité.

2e problème

Comment les autres services peuvent-ils trouver votre document WSDL pour consultation ?

La solution est de centraliser tous les documents WSDL dans un serveur qui tient un annuaire de ceux-ci. Les différents services connaissent l'URL de ce registre. Ils le consultent pour découvrir les nouveaux services et lire les WSDL afin de communiquer ensemble. Ils ont ainsi l'URL du service, ses fonctions et le type de réponse qu'il va renvoyer.

Ce type d'annuaire est appelé UDDI *(Universal Description Discovery and Integration)*.



UDDI pour la découverte de services

3e problème

Comment faire communiquer des services écrits dans des langages différents et utilisant des technologies variées ?

Le XML est la solution, un langage standardisé, extrêmement flexible et personnalisable. SOAP s'est donc imposé comme un protocole qui fixe les règles et les bonnes pratiques pour l'utilisation du XML dans les échanges de messages entre services.

Le service mobile dans notre exemple enverra alors un fichier XML grâce à une requête POST via HTTP vers votre service de gestion des clients. Ce fichier contient des instructions conformes à votre contrat WSDL.

Il vous suffit de parser ce fichier et de renvoyer une réponse SOAP.

4e problème

Comment faire pour créer des messages SOAP conformes afin d'appeler des services ou y répondre ?

Bien que vous devriez comprendre la structure et le fonctionnement des fichiers XML qui répondent aux normes SOAP, vous n'aurez dans les faits jamais à en créer un vous-mêmes !

En effet, la plupart des langages disposent de leurs propres librairies pour vous épargner la peine de formater des fichiers XML complexes à la main.

En Java, la librairie par défaut est JAX-WS. Elle va vous permettre de :

- **Générer les WSDL** : décrire les classes spécifiées dans votre service dans un fichier WSDL à publier, comme vu plus haut, dans un serveur UDDI.
- **Invoquer un service** : il vous permettra de récupérer les fichiers WSDL et d'en générer des Class "proxy" que vous appellerez depuis votre code comme n'importe quelle Class. Jax-WS générera ensuite les messages SOAP et gérera le parsing des réponses.

Chers amis, t’as pas compris ?

Ce n'est pas grave ! Vous allez vous frotter à tous ces concepts et formats de fichiers quand vous créerez un vrai service dans la deuxième partie de ce cours. Tout finira par s'éclaircir !

Résumons

- La SOA définit les concepts qui permettent de mettre en place des SI articulés autour de services et communiquant de façon standardisée.
- Un contrat WSDL est un document XML qui explique les fonctionnalités d’un service, comment l’appeler, où le trouver et quels types de réponses il renvoie.
- UDDI est un serveur qui centralise tous les contrats des services.
- Les services doivent communiquer via SOAP : un protocol XML standard.
- En Java, on utilise des librairies comme JAX-WS pour générer les documents WSDL et les messages SOAP.

PARTIE 2 : Assimilez les 8 commandements de la SOA

Pour vous assurer que votre architecture SOA est fiable et exploiter au maximum son potentiel, voici 8 règles fondamentales à respecter.

Un contrat standard vous appliquerez

Il est important que les documents WSDL soient standardisés au maximum afin de faciliter les réutilisabilités des services.

Cette standardisation implique par exemple des conventions de nommage standardisées, des types de données standards (types complexes qu'on verra plus tard, etc.).

Un couplage faible vous appliquerez

Vos services doivent êtres indépendants et faiblement couplés aux autres composants du SI. Ceci se concrétise en veillant à ce que tous les échanges de votre service avec le monde extérieur se fassent uniquement via SOAP.

Le fait d'aller interroger directement un service, par exemple, crée une dépendance entre ces deux services. Ainsi si demain le service que vous appelez directement change d'adresse ou est supprimé, votre service n'est plus utilisable non plus. Le seul couplage toléré dans une SOA est celui avec un UDDI contenant vos contrats publiés.

Abstraits vos services seront

Les contrats que vous publiez doivent indiquer le minimum vital nécessaire à l'invocation de votre service. En aucun cas vous ne devez publier des informations sur le fonctionnement interne du service.

Toute information non utile directement à l'invocation du service peut être utilisée par d'autres services et générer des réponses inattendues.

Ceci crée également un couplage fort car les autres services du SI seront dépendants de l'implémentation que vous faites.

Par exemple : imaginez que vous exposez une fonction permettant d'avoir la liste complète des clients. Pour avoir cette liste, vous avez une méthode interne qui prend un paramètre "**limit**" qui fixé à -1 retourne la liste complète de tous les clients. Le fait de publier cette fonction interne dans votre WSDL peut tenter d'autres services qui veulent avoir juste les 10 premiers clients à appeler votre fonction interne.

Problème : 1 mois plus tard, vous vous rendez compte que cette fonction peut bien fonctionner sans le paramètre "**limit**" car de toute façon, elle sera toujours amenée à retourner la liste complète des clients, et vous changez donc votre fonction. Ceci aura comme conséquence que tous les services appelant votre fonction interne se retrouvent avec une erreur en guise de réponse à leurs requêtes les empêchant, de ce fait, de fonctionner correctement.

Solution : publiez toujours le strict minimum à l'invocation de votre service.

Des services réutilisables vous développerez

La réutilisabilité d'un service se mesure à son indépendance de la logique de son processus métier.

En d'autres termes, votre service doit retourner les réponses les plus neutres possibles afin que celles-ci soient réutilisables dans différents composants du SI.

Exemple : votre service retourne une liste de produits avec leurs détails : nom, image, description, prix.

Comme vous savez que votre service sera appelé par l'application web qui affichera des promotions pendant la période de Noël, vous pouvez être tenté de retourner le prix final du produit déduction faite des ristournes.

Félicitations, vous venez de manquer au 4e commandement !

Vous avez laissé la logique métier influencer la logique de votre service. Votre service n'est pas réutilisable. Pourquoi ? Imaginez qu'un service qui s'occupe de la vente des produits en gros aux professionnels qui eux ne bénéficient pas des réductions de Noël réservées aux particuliers, souhaite avoir les prix des produits. Votre service n'est pas exploitable dans ce cas car le prix retourné est spécifique à la logique de l'application web et son contexte.

Solution : retournez autant que possible des données neutres. C'est au service appelant d'appliquer sa logique et de manipuler ces données retournées pour les adapter à son cas d'utilisation.

Autonomes vos services seront

Quand vous développez un service web, faites-en sorte qu'il soit le plus autonome possible, c'est-à-dire qu'il dispose de ses propres ressources, libraires, containers, etc.

Un bon service web est un service web que vous pouvez packager dans un **war** ou **jar** par exemple et le faire fonctionner dans n'importe quelle machine de façon indépendante.

Sans état (stateless) vos services seront

Quand vous développez un service, il faut veiller qu'aucune requête ne dépende de la réponse d'une autre.

Exemple :

Votre service gère le panier dans un e-commerce. Vous pouvez être tenté d'implémenter le fonctionnement suivant :

- Un service externe appelle le vôtre en ajoutant un produit A pour le client Robert, puis fait un autre appel pour ajouter un produit B pour le même client.
- Vous gardez ensuite en mémoire une session qui gère le panier de cet utilisateur de façon que quand le client souhaite passer la commande, le service externe vous passe la simple requête "commander". Comme vous avez tout ce qu'il faut en session, il suffit d'enregistrer la commande.

Votre service a donc maintenu une session en **état** pendant une période assez longue, et le service externe comptait sur cet état précis pour passer la commande.

Un des inconvénients de cette approche est que si vous avez 3000 clients sur la boutique, vous saturez la mémoire vive avec 3000 sessions actives. De plus, votre service doit rester actif pour maintenir les sessions ce qui réduit la scalabilité du système : en d'autres termes on aimerait pouvoir ajouter des instances de votre service et en supprimer en fonction du trafic, mais comme votre service a un état, il est difficile de supprimer une de ces instances sans perdre des paniers par exemple.

Solution : quand un service externe demande à ajouter un produit A au client Robert, il suffit d'enregistrer cette donnée dans le système de gestion des données (base de données par exemple) et de retourner l'id du panier. Quand le service externe souhaite ajouter un autre produit ou passer la commande, il est de sa responsabilité de vous passer l'id du panier sur lequel il faut faire l'opération. Ainsi n'importe quelle instance de votre service peut répondre à une telle demande, votre service est donc **stateless**.

Découvrabilité

Comme on l'a vu plus tôt dans le cours, votre service doit pouvoir être trouvé facilement par les autres services grâce à la publication de son contrat (WSDL) dans un annuaire de type UDDI.

Composabilité

Ce commandement est l'aboutissement des 7 commandements précédents. Vous avez créé un service à couplage faible, abstrait, réutilisable, autonome, **stateless** et découvrable. Il est temps de profiter de votre dur labeur !

Votre service doit maintenant participer à la composition d'une application ou d'un SI plus large en proposant ses fonctions à différents composants. La réutilisabilité du service doit être exploitée au maximum afin d'optimiser le système final.

Tout cela peut vous paraître un peu abstrait, mais ne vous inquiétez pas. Vous allez appliquer tous ces commandements quand vous construirez pas à pas un service web dans la 2e partie et ils prendront plus de sens pour vous !

PARTIE 4 : Créez une SOA à partir d'un cas réel

Vous voilà embauché par **ISETTECH** pour faire évoluer leur système vieillissant vers une nouvelle architecture SOA moderne.

Nous n'allons pas faire ici le travail des architectes logiciels mais plutôt essayer de comprendre la logique du découpage d'un système en services.

À quoi tout cela va vous servir ?

Eh bien si vous travaillez sur un SI existant d'une entreprise par exemple, imaginez qu'on vous demande de créer un service de gestion des stocks. Il vous revient de déterminer les limites et les fonctionnalités de votre service pour qu'il réponde aux 8 commandements qu'on a vu précédemment et qu'il soit un service web IRA :

I : Interesting

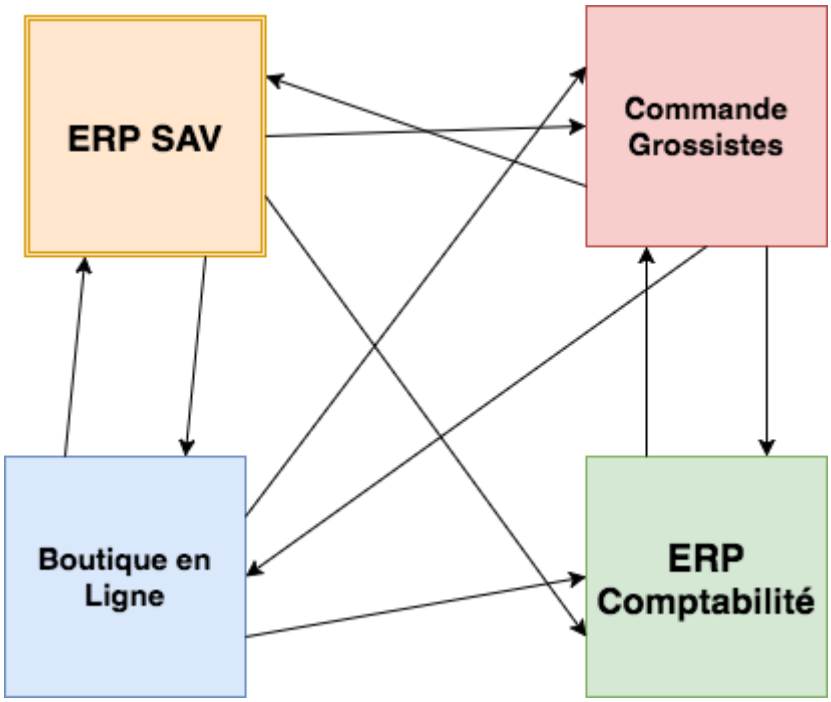
R : Reusable

A : Atomic

Si vous ne savez pas découper les fonctionnalités qu'on vous demande en services dignes de ce nom, vous n'arriverez jamais à l'étape : coder.

Rappel du système de départ

Comme vous vous souvenez (puisque vous suivez assidûment bien sûr), BUZZ dispose d'un système vieillissant dans lequel tous les composants communiquent ensemble directement grâce à une armada d'adaptateurs pour convertir, traduire et reformater les messages entre chaque application.



UDDI pour la découverte de services

Revenons en détail sur les différents composants.

ERP SAV Il s'agit d'un logiciel clé en main propriétaire acheté par l'entreprise. Le protocole de communication de ce logiciel est fixé par le constructeur. Dans ce cas, il s'agit du format JSON via HTTP.

En d'autres termes, si le composant "boutique en ligne" voudrait récupérer les détails d'un incident relatif à un client afin d'afficher le suivi de son traitement à celui-ci, il faut que la boutique en ligne envoie une requête au format JSON, elle recevra également une réponse au même format qu'elle devra comprendre et traiter.

ERP Comptabilité

Ce logiciel est également propriétaire. Le protocole de communication proposé par celui-ci est CSV via FTP.

Boutique en ligne

Il s'agit d'un CMS de type Prestashop qui fournit une boutique clé en main. Le protocole de communication ici est XML via HTTP.

Commandes Grossistes

Ce module est le nouveau web service que vous allez devoir développer.

À ce stade, vous avez dû vous poser la question de comment vous allez transformer ce système pour le baser sur une architecture orientée services si aucun des composants ne communique via SOAP, ne dispose pas de WSDL et en plus chacun a son propre protocole réseau préféré : FTP, HTTP, etc.
Ne vous inquiétez pas, nous allons voir comment nous allons pouvoir harmoniser tout cela grâce à un **ESB**.

Définir les web services

Vous allez d'emblée éliminer : "la boutique en ligne", "ERP comptabilité" et "ERP SAV" de la liste des composants à découper car ce sont des systèmes indivisibles voire propriétaires ou fermés.

Il reste le nouveau venu : le composant des commandes des grossistes.

Étape 1 : Définir les fonctionnalités du composant

La première étape consiste à établir clairement une liste des fonctionnalités qu'on attend du composant à développer, celles-ci nous aideront plus tard à les assembler en services.

Vous serez souvent amené à dresser cette liste de fonctionnalités à partir d'un descriptif du fonctionnement du composant.

Voici le descriptif du module "Commandes Grossistes"

- Le composant devra recevoir des fichiers CSV par FTP dans lesquels des grossistes auront dressés la liste des produits à commander et leurs quantités.

- Ce fichier est ensuite stocké dans un répertoire. Les grossistes doivent pouvoir le mettre à jour ou le supprimer tant que la commande n'a pas été traitée.
- Votre module doit passer à des heures fixes (10h et 16h) pour relever les fichiers et passer les ordres des commandes.
- Il doit ensuite créer un fichier CSV de réponse à mettre dans le même répertoire FTP et qui contient le suivi des commandes : acceptée, traitée, expédiée, annulée, etc.
- Les grossistes reçoivent un email quand leurs commandes passent à certains états : traitée, annulée.

Vous allez maintenant définir une liste de fonctionnalités à partir de ce descriptif :

- Fonction d'upload et de suppression de fichiers de commandes CSV via FTP
- Fonction de validation des fichiers CSV (bon format et contenu)
- Fonction de récupération à heures fixes des commandes
- Fonction de passage des ordres des commandes
- Fonction de suivi des commandes : vérification du statut et mise à jour du CSV de suivi
- Envoi des emails en fonction des statuts des commandes

Maintenant que nos fonctionnalités sont clairement définies, on va pouvoir créer un découpage des web services, le plus optimisé possible.

***Étape 2* : Découper le composant en web services**

Découper un composant logiciel en web services est une tâche bien plus délicate que ce que l'on peut penser.

Dans notre cas, plusieurs solutions sont possibles. La plus intuitive serait :

- Un service qui s'occupe de tout ce qui est opérations sur les CSV : upload, validation, etc. ;
- Un service qui s'occupe des commandes.

Ceci paraît clair et propre, mais ce genre de découpage est exactement ce qu'il faut éviter.

Pour vous aider à découper vos web services de manière à ce qu'ils soient compatibles avec une SOA, il faut respecter les 2 règles suivantes :

- Votre web service doit être IRA.
- Votre web service doit respecter les 8 commandements.

Qu’est-ce qu’un web services IRA ?

I : Interesting

R : Reusable

A : Atomic

Interesting :

Les fonctionnalités de votre web service doivent être intéressantes non seulement pour servir la finalité du composant que vous développez mais potentiellement intéressantes pour l’ensemble des web services du système.

Exemple : Un service qui met à jour un champ dans une table de base de données dès qu’une commande est ajoutée est totalement inintéressant, car il s’agit d’un service qui traite du fonctionnement interne du processus de commande (par exemple dans la boutique en ligne). Aucun autre web service ne sera intéressé par la mise à jour d’un champ obscur dans une table aux tréfonds de la base de données d’un CMS !

Reusable :

Votre web service doit être réutilisable dans d’autres contextes et scénarios. Il ne doit pas être lié à la logique du processus dans lequel il intervient.

Exemple : un web service qui renvoie les 10 derniers produits ajoutés au stock n’est intéressant que dans un cadre précis (par exemple pour afficher les nouveautés dans une page de la boutique), alors qu’un web service qui renvoie n’importe quel nombre de produits qu’on lui demande est potentiellement intéressant pour beaucoup d’autres web services. Par exemple l’ERP Comptabilité peut faire appel à ce web service pour récupérer l’ensemble des prix des produits dans le catalogue !

Atomic :

Faites une seule chose et faites-la bien ! Atomic veut dire indivisible et élémentaire. Évitez les fonctions dans vos web services qui par exemple passent les commandes, gèrent les retours, suppriment les produits et font le café. Évitez aussi une trop grande atomicité, on ne va pas créer un web service pour récupérer le nom d’un produit et l’autre sa description par exemple. Le tout est que celui-ci soit un bloc naturel et élémentaire.

Exemple : Un service qui a une fonction qui renvoie toutes les informations sur tous les grossistes et leurs commandes depuis la création de leurs comptes. Ce web service n'est clairement pas atomic : il sera composé d’une dizaine de composants qui iront chercher les infos des grossistes, les commandes, les prix, les infos sur les produits de chaque commande, etc. Un web service atomic serait plutôt celui qui aura des fonctions séparées : une pour renvoyer la liste des grossistes, une autre la liste des commandes, etc. La logique étant que si nous voulons toutes ces informations, il suffit d’appeler chacune de ces fonctions et d'agréger les résultats.

Challenge 5 min Appliquez ce que vous avez appris plus haut pour réaliser un découpage propre de notre module de Commandes de Grossistes.

Assurez-vous que vos services répondent aux règles, puis revenez pour les comparer à mon découpage !
Maintenant que vous avez pris le temps de réfléchir au problème, voici une proposition de découpage.

Service 1

Reçoit un fichier en entrée et l'upload dans un répertoire. Il peut également supprimer un fichier ou le remplacer.

Intéressant ? Bien sûr, beaucoup de composants peuvent être intéressés par la récupération et l'upload de fichiers divers et variés.

Réutilisable ? Absolument. Il n'est lié à aucune logique en particulier ni à aucun format de fichier, n'importe quel composant peut le réutiliser.

Atomic Oui. Le service ne fait qu'uploader des fichiers, c'est une fonctionnalité tout à fait fondamentale et indivisible.

Service 2

Reçoit des fichiers dans des formats populaires (CSV, XML, etc.) et les valide.

Intéressant ? Tout à fait. La plupart des composants sont amenés à utiliser des fichiers de ce type et doivent les valider.

Réutilisable ? Bien sûr. Un service qui centralise la validation des fichiers selon des critères établis à l'avance, sera appelé par tous les composants au besoin et évitera que chacun réécrive la logique de validation de chaque fichier.

Atomic Oui. La validation est un processus indivisible.

Service 3

1. Reçoit en entrée un fichier CSV puis place les ordres de commandes.
2. Reçoit en entrée une information de mise à jour de commande, puis met à jour le CSV de suivi des commandes.

NOTE Ici on a un web service qui offre plusieurs fonctionnalités. C'est souvent ce genre de web service que vous allez rencontrer. Les règles dans ce cas s'appliquent fonction par fonction.

Intéressant ? Oui. Avoir un moyen de placer des commandes et d'assurer leur suivi via un simple fichier CSV pourrait intéresser plusieurs composants et entrer dans plusieurs cas d'utilisation.

Réutilisable ? Oui. *Fonction 1* : Plusieurs composants pourraient se servir de ce service pour placer des commandes très simplement via CSV. Par exemple, un service qui s'occupe des ventes privées ou groupées, pourrait l'utiliser pour passer les commandes sans avoir à recréer l'ensemble de ce que nous mettons en place pour les commandes des grossistes.

Fonction 2 : Absolument ! Tous les composants du système qui entrent dans le traitement de commande feront appel à cette fonction pour mettre à jour le statut des commandes. Exemple : les systèmes de paiement, de gestion de stock, d'expéditions, de retours, etc.

Atomic Oui. Le processus de passage de commande est un processus cohérent, extrêmement interconnecté et qui se fait en un seul bloc.

Vous aurez donc compris la logique de découpage, je vous donne les 2 derniers services nécessaires et je vous laisse vérifier s'ils sont conformes aux règles :

- ***Service 4*** : Un service de mailing qui sera appelé par le service de commande pour envoyer des emails aux grossistes pour les notifier de l'évolution de leurs commandes. Bien entendu, ce service peut être utilisé par n'importe quel autre composant pour envoyer des emails.
- ***Service 5*** : Un service qui implémente une fonction de type CRON, qui pourra appeler à intervalles réguliers d'autres services. Dans notre cas, il servira à appeler le service 3 pour qu'il récupère les CSV à des heures précises.

Cependant, tout cela ne sert à rien si vos services ne peuvent pas communiquer avec les autres composants du SI. Comment allez-vous faire si vous voulez passer les détails d'une commande à l'ERP comptabilité pour prise en compte ? Ou si vous voulez demander à la boutique en ligne s'il n'y a pas des commandes en attente sur un stock pour pouvoir le vendre de votre côté à un grossiste qui en a fait la demande ?

Eh oui ! Je vous rappelle que vos services ne parlent pas encore la même langue ! Regardons comment remédier à cela.

Faire communiquer les composant "Non SOAP" grâce à un ESB

Nous allons dans cette section survoler les concepts d'un ESB sans pour autant entrer dans les détails. Votre but premier dans ce cours est de savoir créer des services compatibles avec une architecture SOA et les faire communiquer ensemble. Le reste relève plus du travail d'un architecte logiciel !

Dans la prochaine partie on écrira du code et vous allez voir, ça ira mieux !

Quel est notre problème ?

Notre problème réside dans le fait que nous avons un système incluant plusieurs composants (logiciels, services, etc.) qui ne communiquent pas via les mêmes protocoles, qui ne sont pas écrits dans les mêmes langages et qui sont malgré tout extrêmement dépendants les uns des autres.

Une solution serait d'écrire des bouts de code qui traduisent les messages de chaque composant en quelque chose de compréhensible par le composant à appeler.

Par exemple, si vous voulez que votre **service 3** envoie à l'ERP comptabilité le résumé de chaque commande afin que les mouvements rentrent dans les comptes, il vous faudra communiquer avec lui via CSV/FTP. Or votre service communique

uniquement en SOAP (via des fichiers XML). Donc la solution évidente est d'écrire du code qu'on appellera "adaptateur" qui générera un format CSV plutôt que XML. Puis vous l'enverrez via FTP à l'ERP en question.

voilà qui est résolu,
Malheureusement ce n'est pas si facile que ça. Imaginez maintenant que vous êtes dans un SI plus grand et que vous devez communiquer avec une centaine de logiciels et services. Allez-vous écrire une centaine d'adaptateurs ? Les autres logiciels devront en faire autant ? Qu'en est-il des logiciels propriétaires qu'on ne peut pas modifier ? Vous allez certainement créer d'autres adaptateurs de votre côté pour comprendre leurs messages, par exemple pour interpréter une requête via CSV reçue de l'ERP compta.

L'ESB à la rescousse

Un Enterprise Service Bus (ESB) est un composant central qui se positionnera comme un interlocuteur unique pour tous les composants du SI. Ainsi, quand vous voudrez appeler un composant X, vous n'irez plus jamais lui parler directement. Vous ne saurez même pas quel protocole il accepte, ni son URL ! Il suffira d'envoyer une requête dans votre protocole préféré (dans notre cas SOAP) à l'ESB. Celui-ci s'occupera ensuite de faire le nécessaire pour transformer votre message et l'adapter avant de le transférer au composant demandé. Il fera ensuite la même chose pour la réponse avant de vous la servir sur un plateau d'argent, reste plus qu'à déguster !

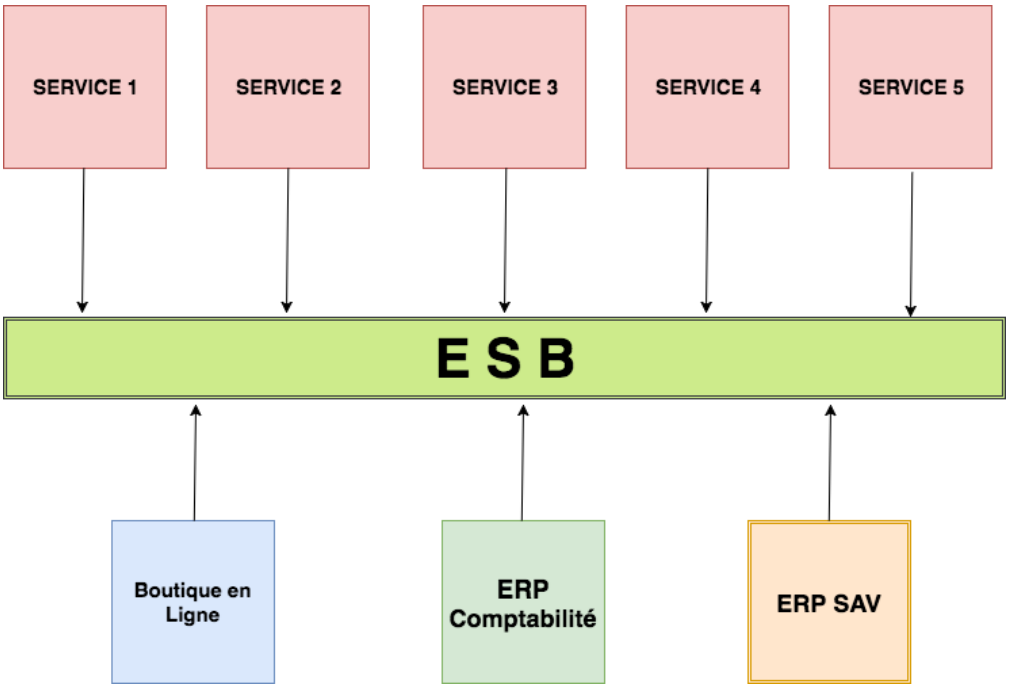
l'ESB n'a pas de pouvoirs magiques, il faudra bien lui fournir des adaptateurs ?

Il faudra bien écrire des adaptateurs à fournir à l'ESB afin que celui-ci sache comment traduire les messages.

Mais l'énorme avantage ici, c'est qu'on n'aura plus une architecture complexe. Vos adaptateurs seront tous centralisés au même endroit. De plus vous n'avez plus besoin de dupliquer vos adaptateurs entre différents composants.

Par exemple, dans l'exemple précédent, si 15 autres services ont besoin aussi de communiquer avec l'ERP Comptabilité, vous allez tout simplement leur fournir votre adaptateur. Vous vous retrouvez avec 10 copies de votre code éparpillées dans le système. Imaginez qu'il y ait un bug à résoudre ? Aïe ! Il faudra appeler tous les développeurs responsables des différents services pour leur demander de faire les changements. Certains d'entre eux seront peut-être obligés d'arrêter le service pour intégrer votre modification, mettant en péril le fonctionnement de tout le SI. Pas joli joli tout ça.

Maintenant que vous mesurez l'importance d'un ESB, regardons à quoi ressemblera notre SI.



Implémentation d'un SOA grâce à un ESB

Vous sentez déjà que c'est plus propre et carré.

Maintenant quand vous écrivez un adaptateur, il n'en existera qu'une seule copie placée dans l'ESB. Vous pouvez le modifier et l'améliorer à volonté en toute transparence.

Pour compléter notre **survol** de l'ESB, voici les avantages qu'il offre :

- **Couplage faible** : les composants de votre SI n'ont plus à se connaître pour communiquer. Chaque service ne communique plus qu'avec l'ESB. Vous pouvez donc remplacer à votre guise un service par un autre, et gagner en liberté dans le choix des langages et des environnements.
- **Standardisation** : quand vous créez un nouveau service, il n'est plus nécessaire de passer des semaines à l'intégrer et le faire communiquer avec les composants du SI. L'intégration est très simple et se fait uniquement avec l'ESB. Ceci va aider également l'entreprise à établir des standards dans les développements des services vu qu'il n'y a plus de contraintes d'intégration. Tous les services à terme se ressembleront dans leurs structures et modes de fonctionnement, ce qui simplifie grandement la maintenance, l'interchangeabilité des postes des développeurs et bien sûr le coût !
- **Scalabilité** : il est maintenant beaucoup plus facile d'avoir plusieurs instances de votre service pour répondre par exemple à un trop grand nombre de requêtes (période de Noël pour **ISETTECH** !). Ainsi, si votre service est en surcharge et ne répond plus, l'ESB pourra rediriger les requêtes vers une autre instance de celui-ci permettant ainsi au système de continuer à fonctionner sans ralentissement ou interruption.

- **Routage** : l'ESB va vous permettre de rediriger les requêtes très finement en fonction des paramètres de votre choix. Imaginez que vous avez décidé de créer une V2 de votre service avec des améliorations significatives qui impliquent des changements dans la façon de l'appeler. Grâce à l'ESB, vous allez rediriger les requêtes arrivant à votre service vers la bonne version de celui-ci selon si elles sont compatibles V1 ou V2.

L'ESB offre de nombreux autres avantages comme ceux liés au messaging, aux protocoles non standards (autres que HTTP), etc. Néanmoins ceux-ci nécessiteront la maîtrise d'autres concepts qui ne font pas partie de ce cours.

Résumons

- Quand vous découpez un composant en service, veillez à ce que chaque service soit IRA et respecte les 8 commandements.
- L'ESB est un composant central qui va se positionner comme un interlocuteur unique pour tous les autres composants, éliminant de fait les problèmes de communication et donnant au système une vraie consistance.

Test : Les principes de la SOA

Compétences évaluées

Concevoir et réaliser l'architecture d'un SI dans un environnement client-serveur

Remarque : pour votre bien éviter le plagiat

Question 1

Quelle est la différence entre une SOA et des web services ?

Question 2

Qu'est-ce qu'un UDDI ?

Question 3

Que fait un service stateless ?

Question 4

Quelle est la caractéristique d'un service Atomic ?

Question 5

Qu'est-ce qu'un ESB ?

PARTIE 5 : Mettez en place l'environnement de développement

Très bien nous y voilà ! Nous allons créer notre premier web service concret !

Pour illustrer ce cours, nous allons créer un service que vous pouvez trouver fun ou glauque en fonction de votre âge. Il s'agit d'un web service auquel on donnera un prénom, un sexe (homme/femme) et une année de naissance et qui va nous retourner le nombre d'années qu'il nous reste à vivre en se basant sur l'espérance de vie moyenne en France.

Pour cela, nous aurons besoin de :

- un IDE : j'ai choisi ici IntelliJ IDEA ;
- un serveur d'application Java EE : Glassfish 5 ;
- le JDK Java 8.

Passons à l'installation.

Installer IntelliJ IDEA

IntelliJ IDEA est un des IDE Java les plus modernes. Il propose un système d'autocomplétion parmi les plus aboutis du marché, ainsi que des fonctionnalités avancées de refactoring et une interface utilisateur offrant une très bonne expérience. Vous pouvez bien sûr utiliser l'IDE de votre choix, je vais m'efforcer de ne pas utiliser de fonctionnalités spécifiques à IntelliJ.

Trêve de prosélytisme ! Passons à l'installation. Si IntelliJ (ou un autre IDE) est déjà installé sur votre machine, vous pouvez bien sûr passer cette section.

Téléchargez IntelliJ IDEA Community Edition ici. Installez l'IDE en suivant les instructions à l'écran.

À l'écran de choix d'import de configuration, cliquez sur ***Do not import settings***.

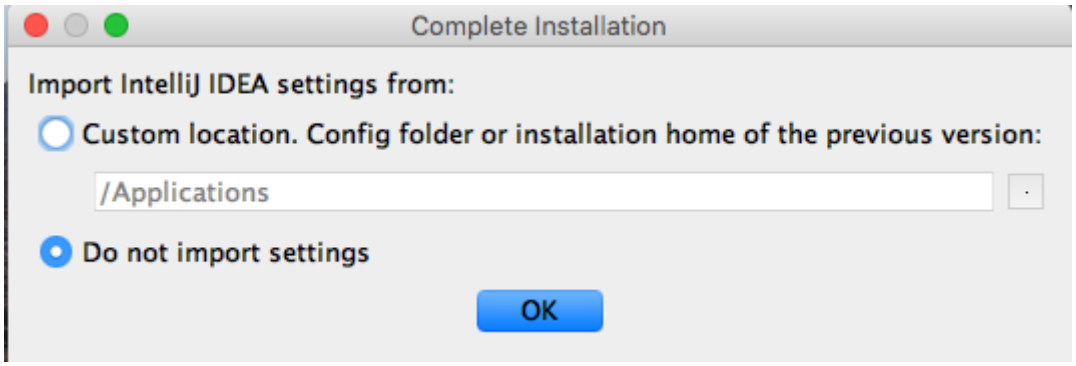


Figure 1

Acceptez les conditions d'utilisation.

Passez à l'écran "Customize IntelliJ IDEA" en cliquant sur ***Skip All and Set Defaults***.

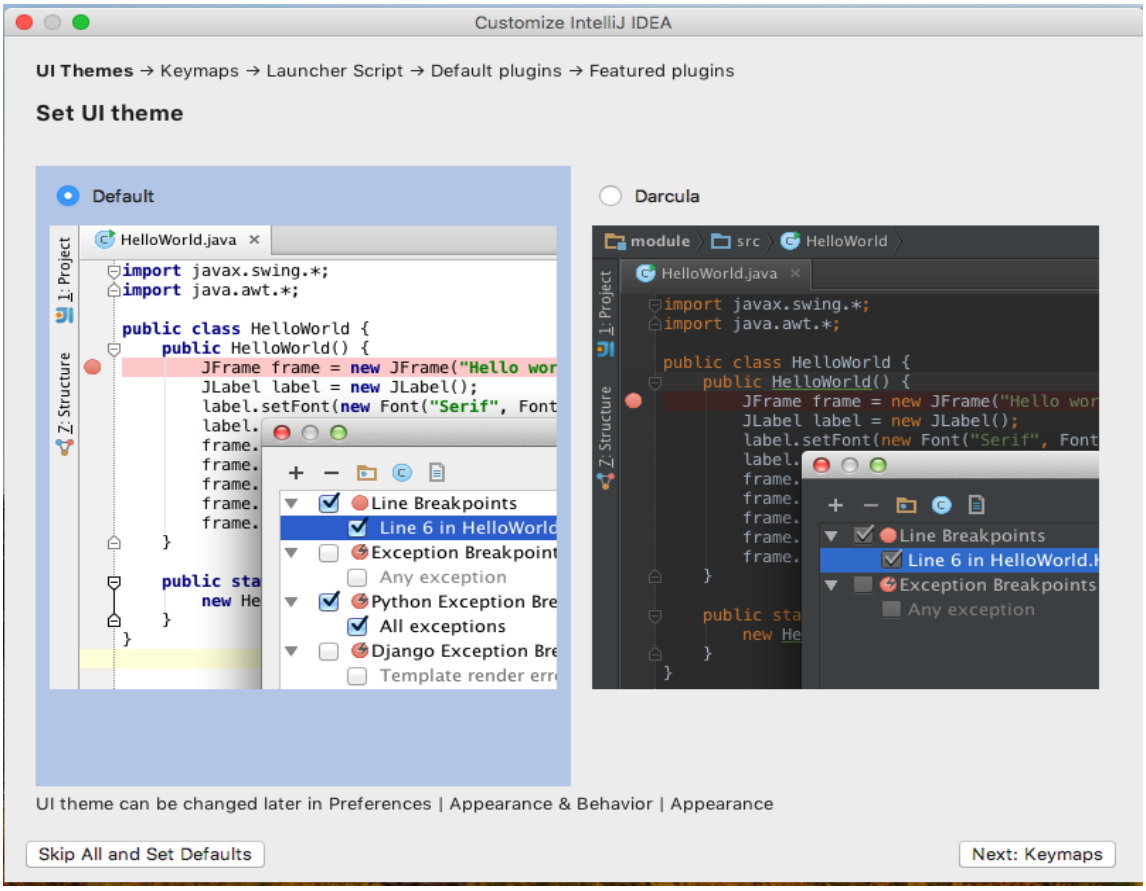


Figure 2

Vous tomberez alors sur l'écran principal de l'IDE.

Figure 3

Cette étape est facultative, mais je trouve que travailler devant l'écran d'un IDE sur fond blanc revient à regarder une ampoule allumée 8 heures par jour. Voici comment changer le thème.

Sous macOS Cliquez sur IntelliJ IDEA en haut à gauche puis *Préférences*.

Sous Windows Cliquez sur *File* puis *Settings*.

Allez ensuite à *Appearance and Behavior* > *Appearance*.

Changez le thème de Default à **Dracula** puis validez.

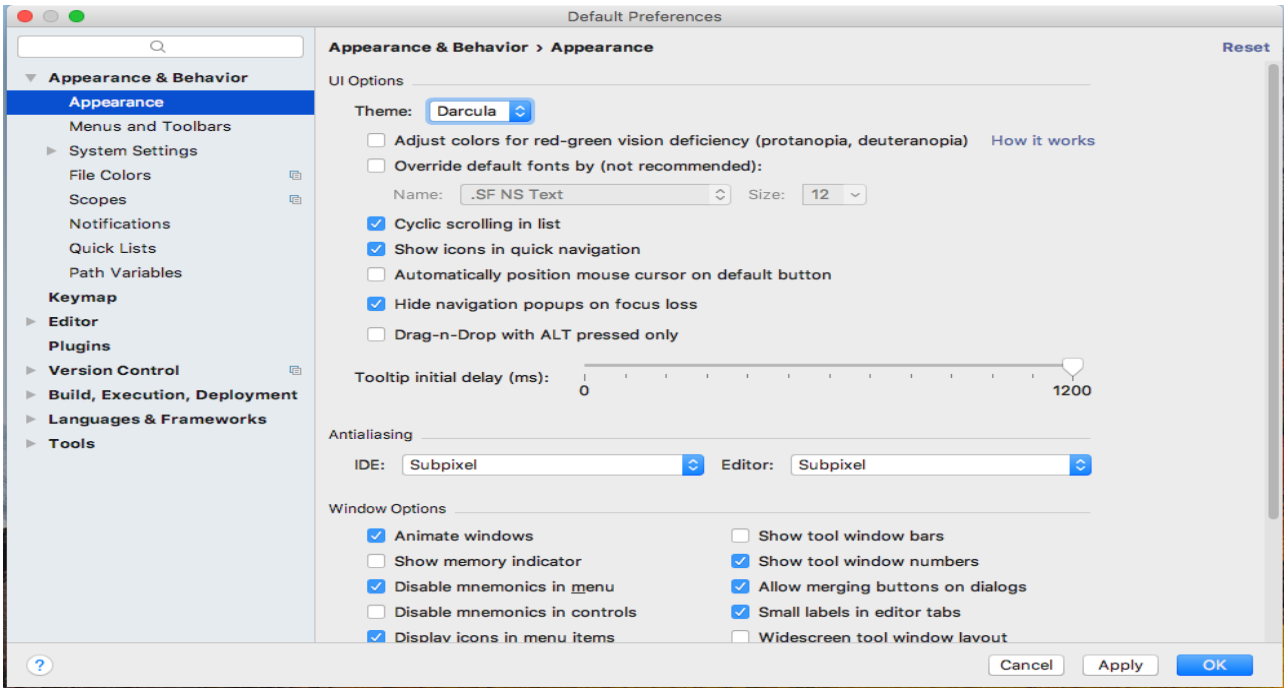


Figure 4

Créons maintenant notre premier projet. Cliquez dans l'écran principal (figure 3) sur *Create New Project*. Vous allez tomber sur un écran affichant différentes possibilités pour créer un projet en partant d'une base. Dans notre cas nous allons partir d'un projet Maven.

Sélectionnez ***Maven*** à gauche, puis cochez la case *Create from archetype*. Sélectionnez ensuite *org.apache.maven.archetypes:maven-archetype-webapp*.

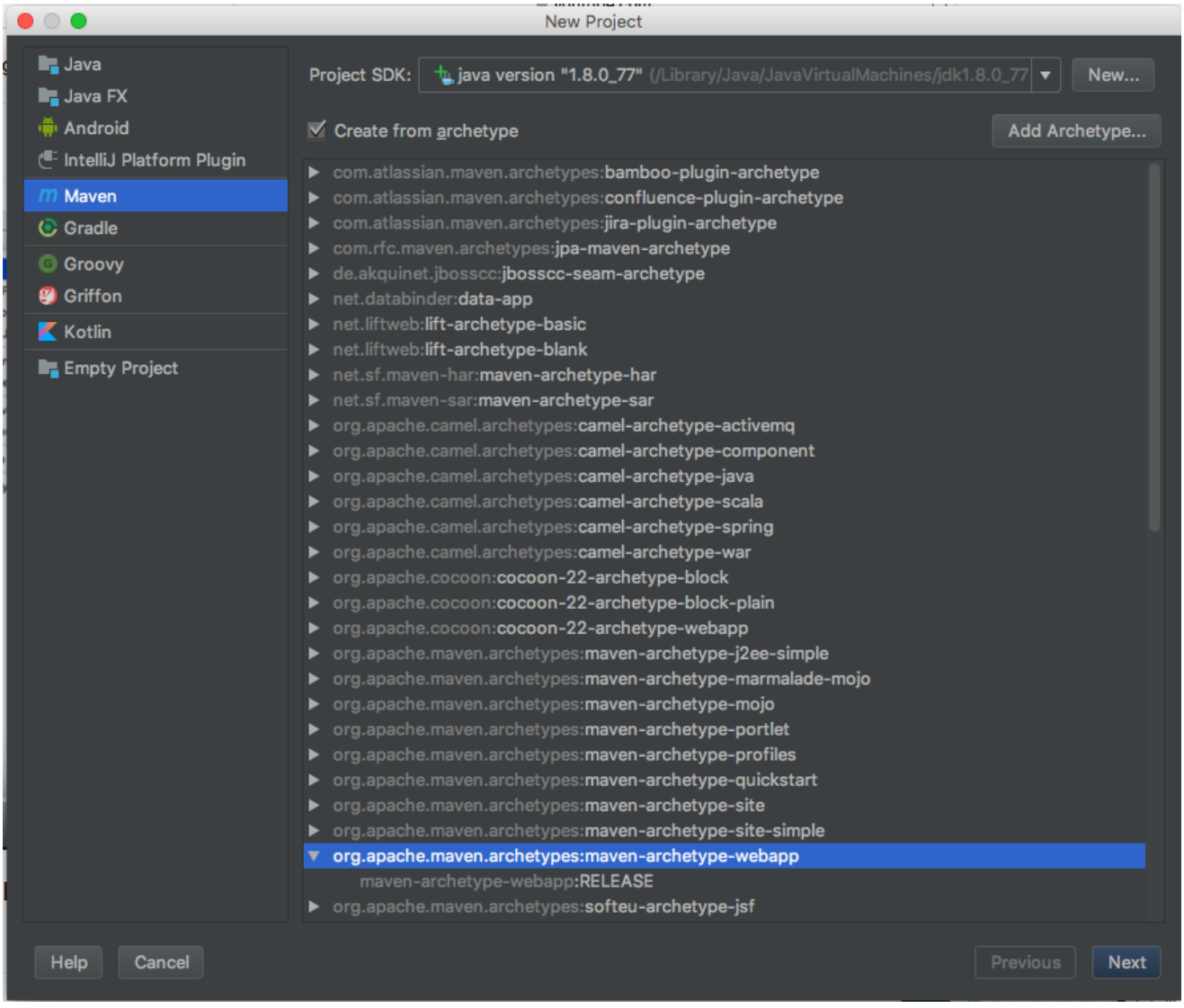


Figure 5

IntelliJ va sélectionner automatiquement le SDK Java installé dans votre machine, il apparaît en haut dans *Project SDK*. S'il n'apparaît pas, vérifiez que vous l'avez dans votre machine en utilisant la commande suivante dans le Terminal si vous êtes sous macOS ou dans cmd.exe sous Windows :

```
java -version
```

Si le SDK est installé, vous devriez recevoir un message comme celui-ci :

```
java version "1.8.0_77"
```

```
Java(TM) SE Runtime Environment (build 1.8.0_77-b03)

Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode)
```

Le SDK n'est pas installé ? Alors je vous invite à télécharger le SDK puis à l'installer en cliquant sur le .exe sous Windows ou en le dézippant tout simplement à l'emplacement de votre choix sous macOS.

Notez bien l'emplacement de l'installation, puis revenez à votre IDE (Figure 5) et cliquez sur *new* puis pointez vers le dossier *jdk1.8.(une certaine version) > content > HOME*.

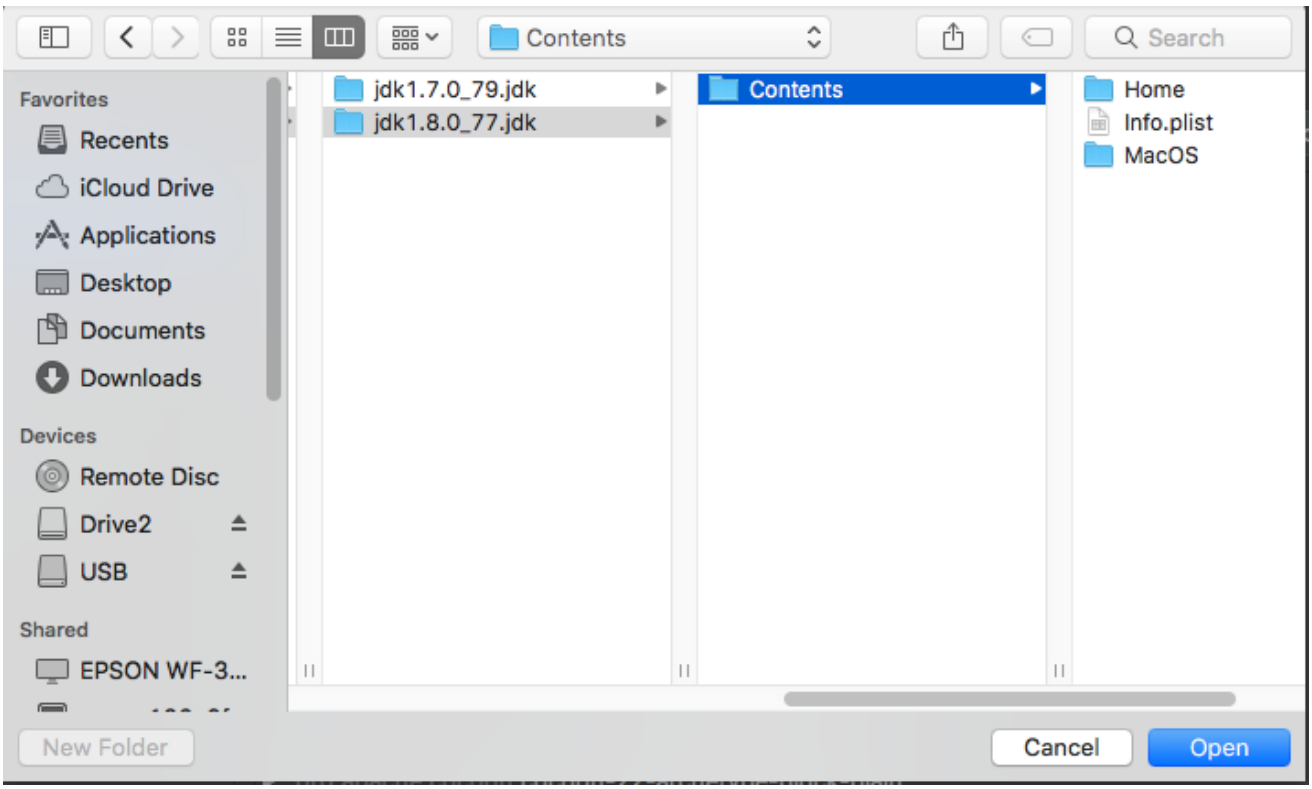


Figure 6

Cliquez sur *Next* dans la figure 5.

Nous allons nommer notre projet **LifeLeft** (parce qu'en anglais ça claque). Remplissez le GroupeID et l'artefactId (respectivement *com.lifeleft* et *lifeleft*), puis cliquez sur *Next*.

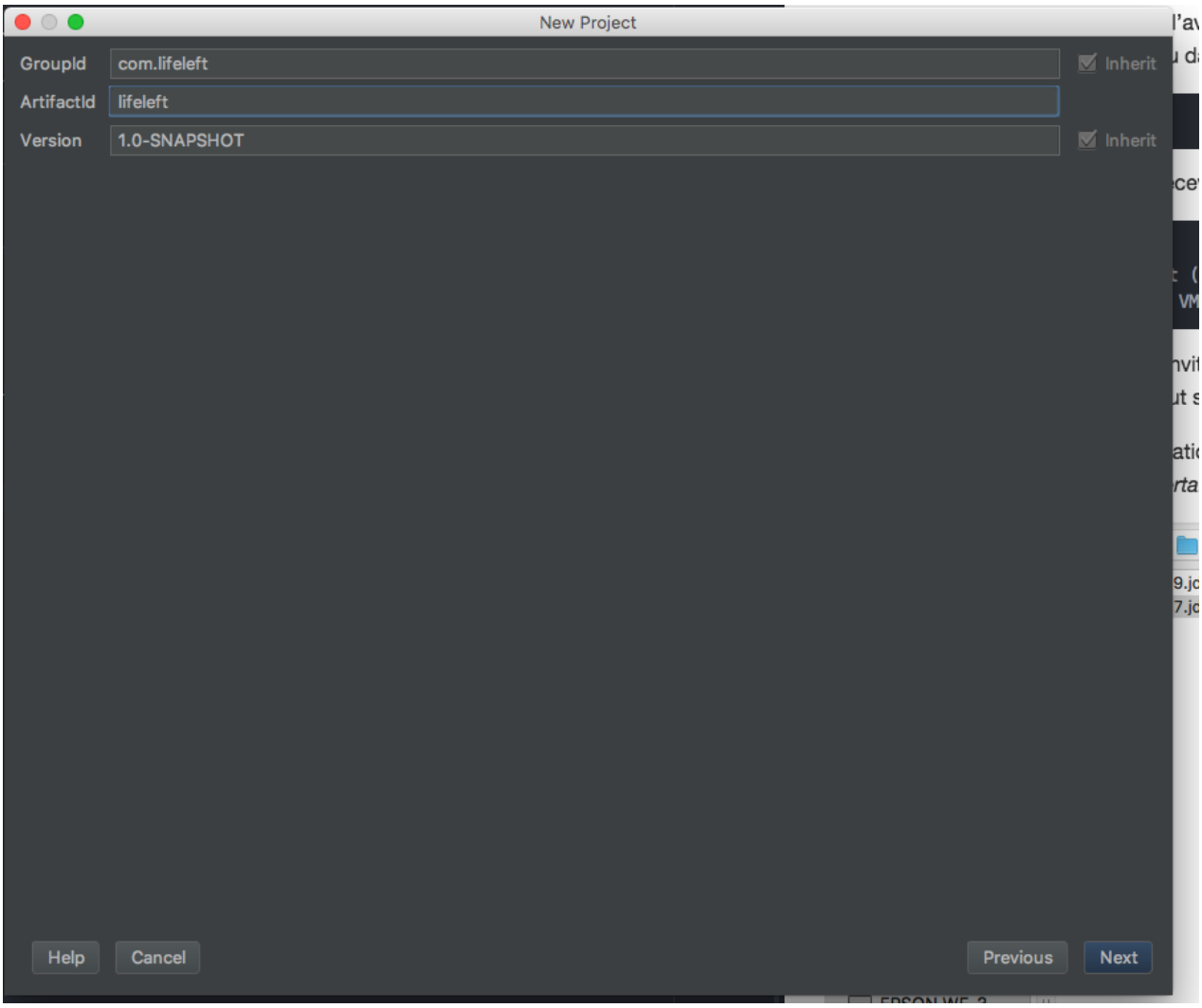


Figure 7

Dans cet écran, vous pouvez personnaliser certaines configurations Maven. Laissez les choix par défaut. Cliquez sur *Next*.

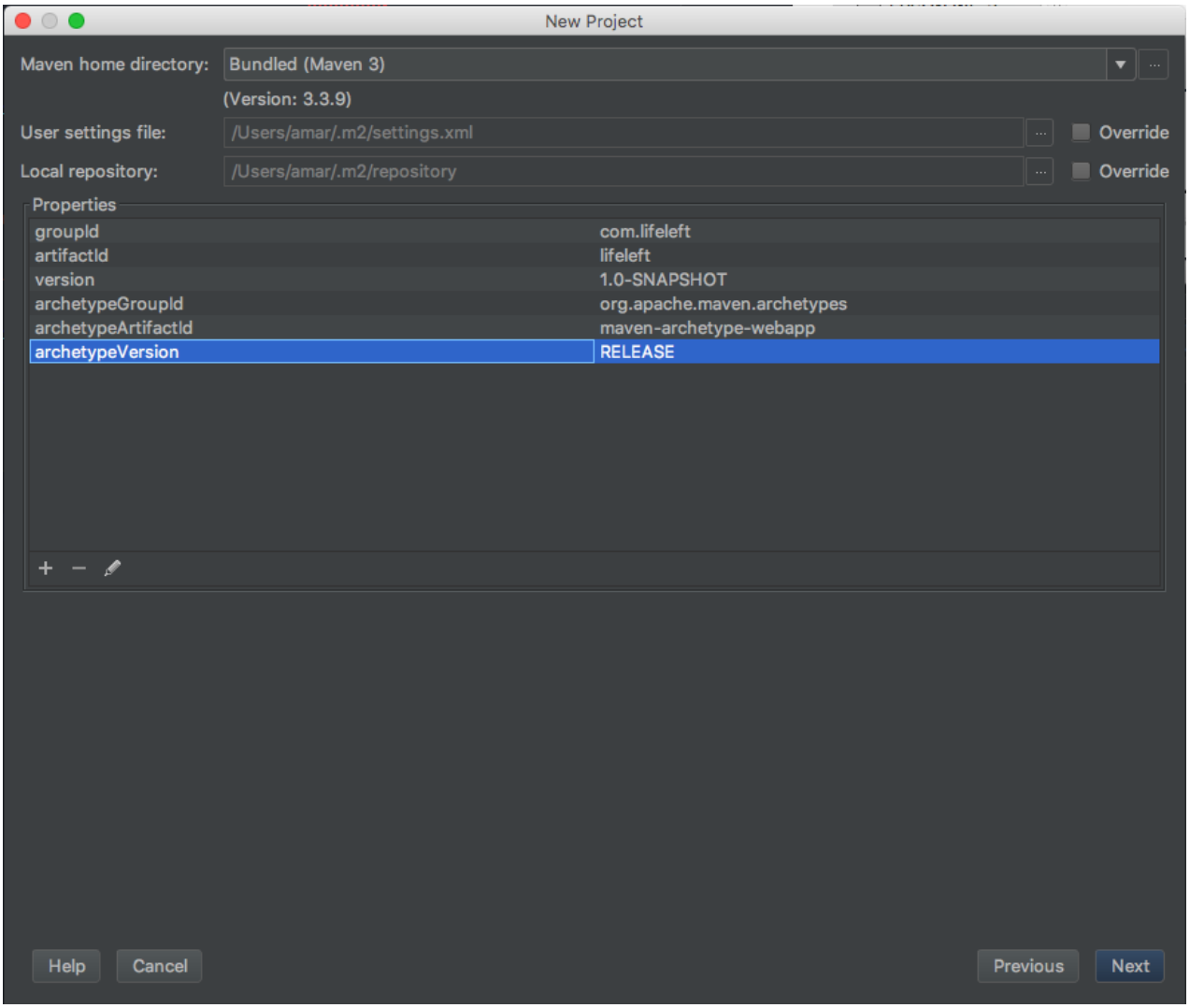


Figure 8

Nommez le projet *lifeleft* puis cliquez sur *Finish*.

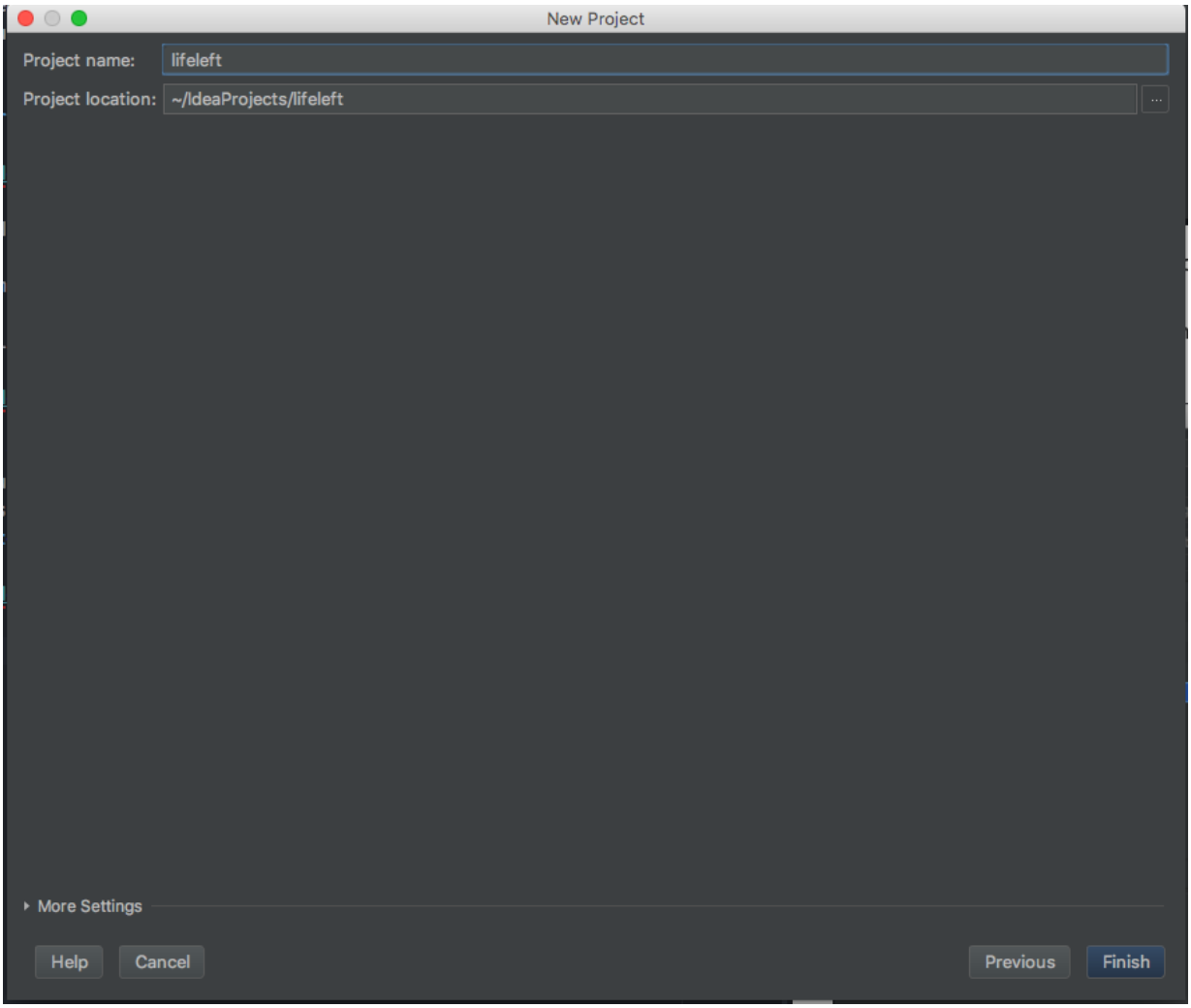


Figure 9

IntelliJ va commencer à télécharger les dépendances requises pour un project Maven de type webapp. Attendez la fin des opérations.

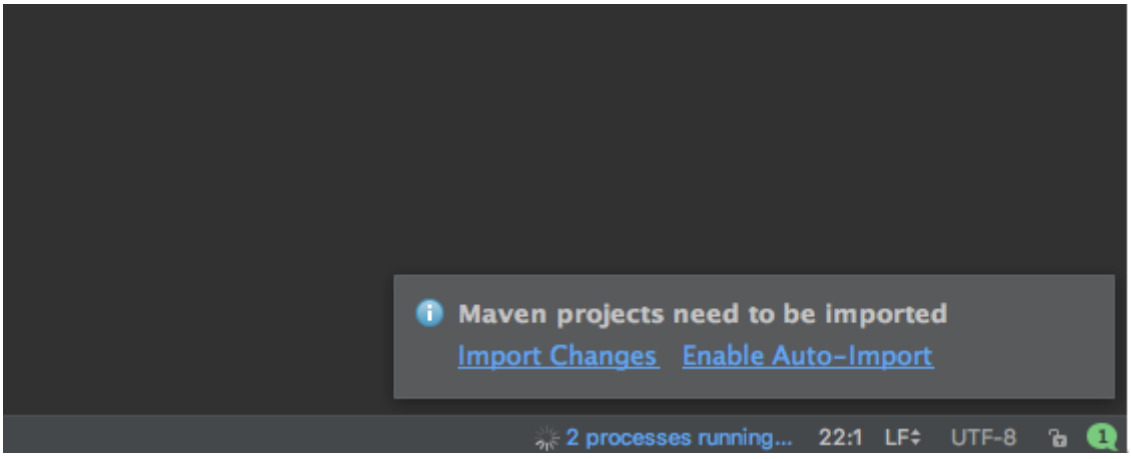


Figure 10

Quand c'est fini, vous devriez avoir cette structure de projet dans le panneau de gauche.

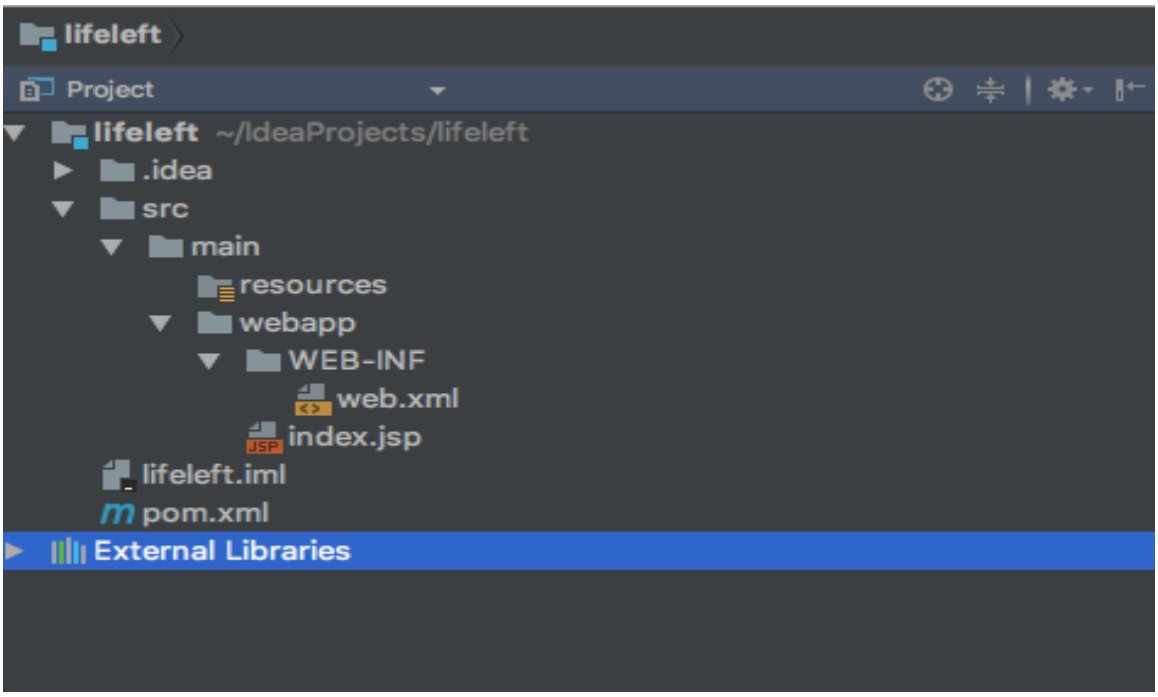


Figure 11

Installer Glassfish

Glassfish est un serveur d'application compatible avec Java EE. Ne vous embrouillez pas, c'est tout simplement l'équivalent de Tomcat pour JEE. Il va nous permettre de travailler facilement avec les web services en offrant beaucoup de fonctionnalités spécifiques à ces développements, comme la possibilité de tester notre web service avec un simple paramètre dans l'URL. Nous y reviendrons.

Téléchargez Glassfish 5.

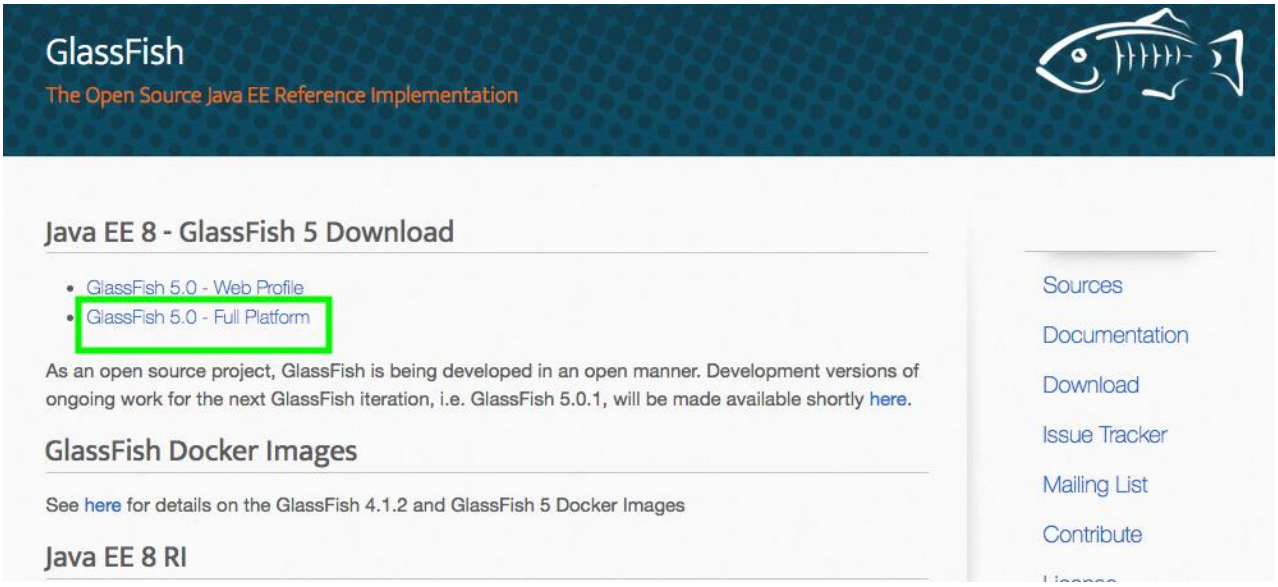


Figure 12

Décompressez le dossier obtenu.

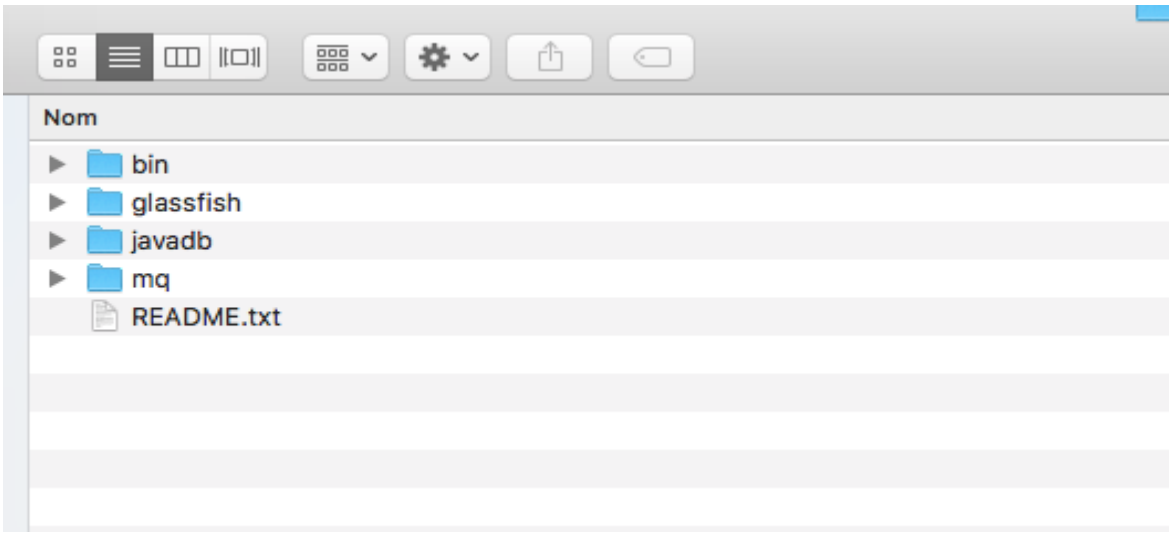


Figure 13

C'est tout ! Vous pouvez dès à présent lancer le serveur en exécutant la commande suivante.

```
/Chemin/vers/dossier/glassfish/bin/asadmin start-domain domain1
```

Vous indiquez tout simplement le chemin complet vers le binaire *asadmin* suivi de start-domain domain1. On obtient donc ceci :

```
MacBook-Pro-de-amar:Downloads amar$ /Users/amar/Downloads/glassfish5/bin/asadmin s
tart-domain domain1
Waiting for domain1 to start .....
Successfully started the domain : domain1
domain  Location: /Users/amar/Downloads/glassfish5/glassfish/domains/domain1
Log File: /Users/amar/Downloads/glassfish5/glassfish/domains/domain1/logs/server.l
og
Admin Port: 4848
Command start-domain executed successfully.
```

Figure 14

start-domain ? domain1 ?

Je ne vais pas trop m'étaler sur la signification du concept de domaine dans Glassfish vu que ce n'est pas le but de notre cours. Mais pour faire très simple, imaginez que le domaine est simplement le nom de votre serveur glassfish. Le domaine par défaut est domain1

Revenons à nos moutons ou plutôt à nos poissons (glassfish... poisson... OK j'arrête !).

Maintenant vous avez un super serveur Glassfish prêt. Le port par défaut du panneau d'administration est 4848.

Rendez-vous donc à ***http://localhost:4848***

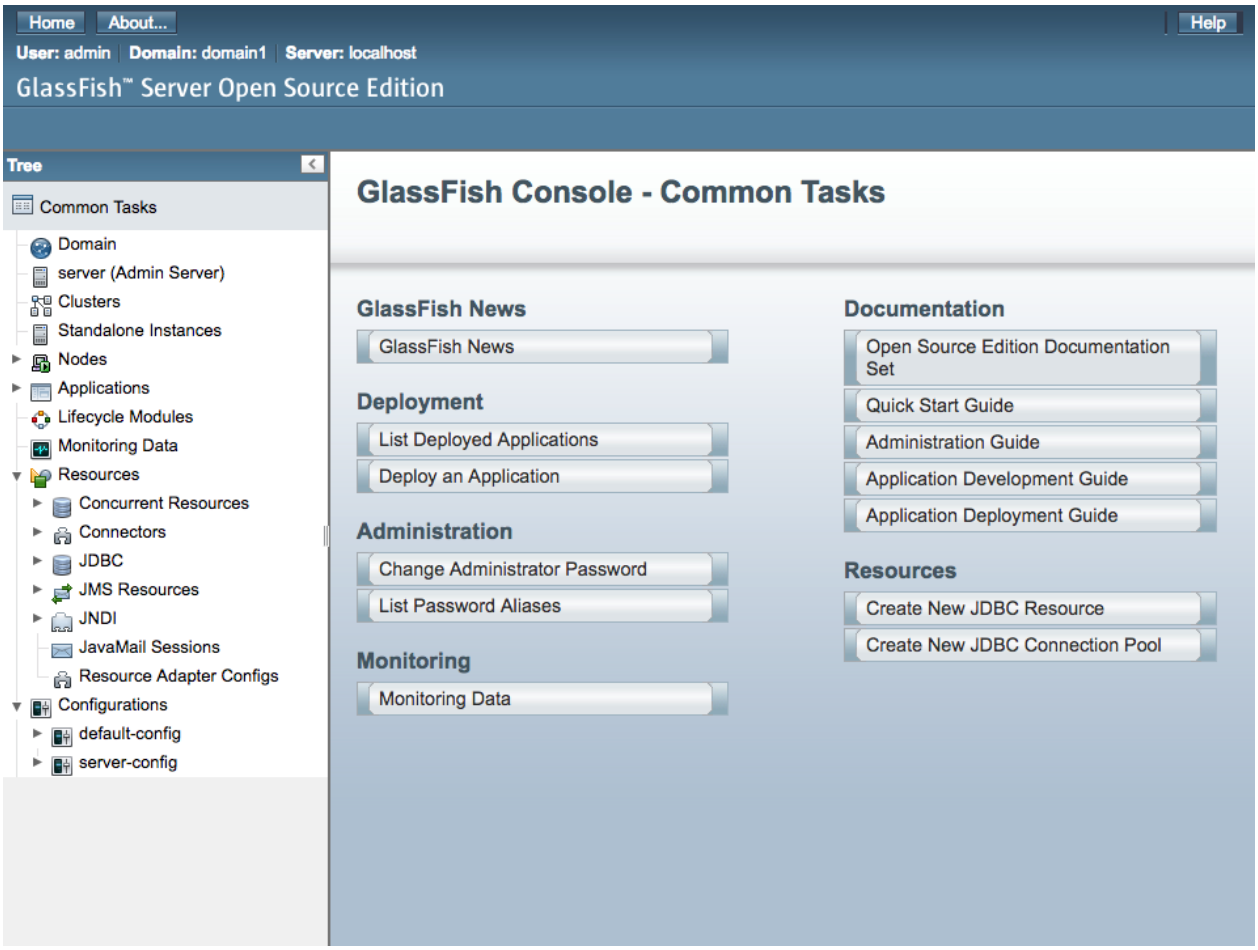


Figure 15

Si vous n'arrivez pas à afficher cette page, vérifiez le port utilisé par Glassfish dans le résultat de l'exécution de la commande plus haut.

Créez un dossier que vous appellerez java sous le répertoire *main*. Pour ce faire, faites un clic droit sur le dossier *main* puis *New* > *Directory* et entrez *java* dans la boîte de dialogue.

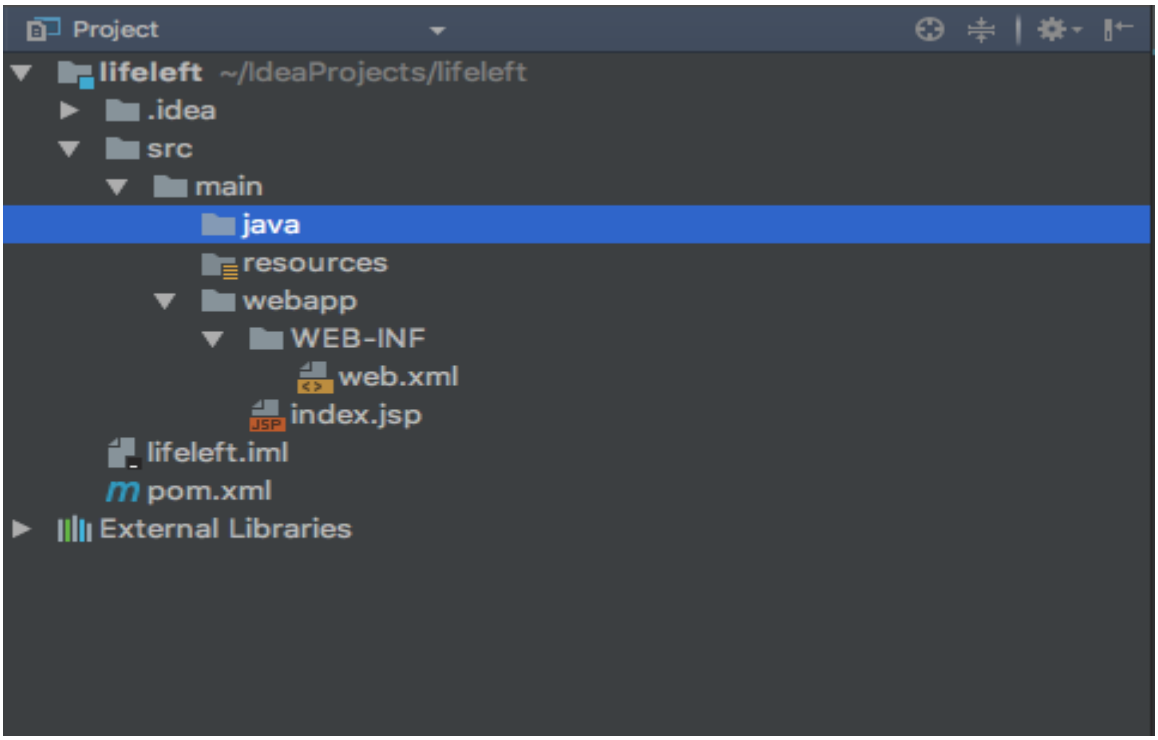


Figure 16

Cliquez ensuite sur *File > Project Structure* puis cliquez à gauche sur *Modules*. Nous allons ici montrer à IntelliJ où est-ce que nous allons mettre notre code Java. Pour cela, nous allons marquer le dossier java comme étant la source. Sélectionnez le dossier java dans le panneau du milieu puis cliquez en haut sur le bouton bleu *Sources*, le dossier *java* passe au bleu. Validez.

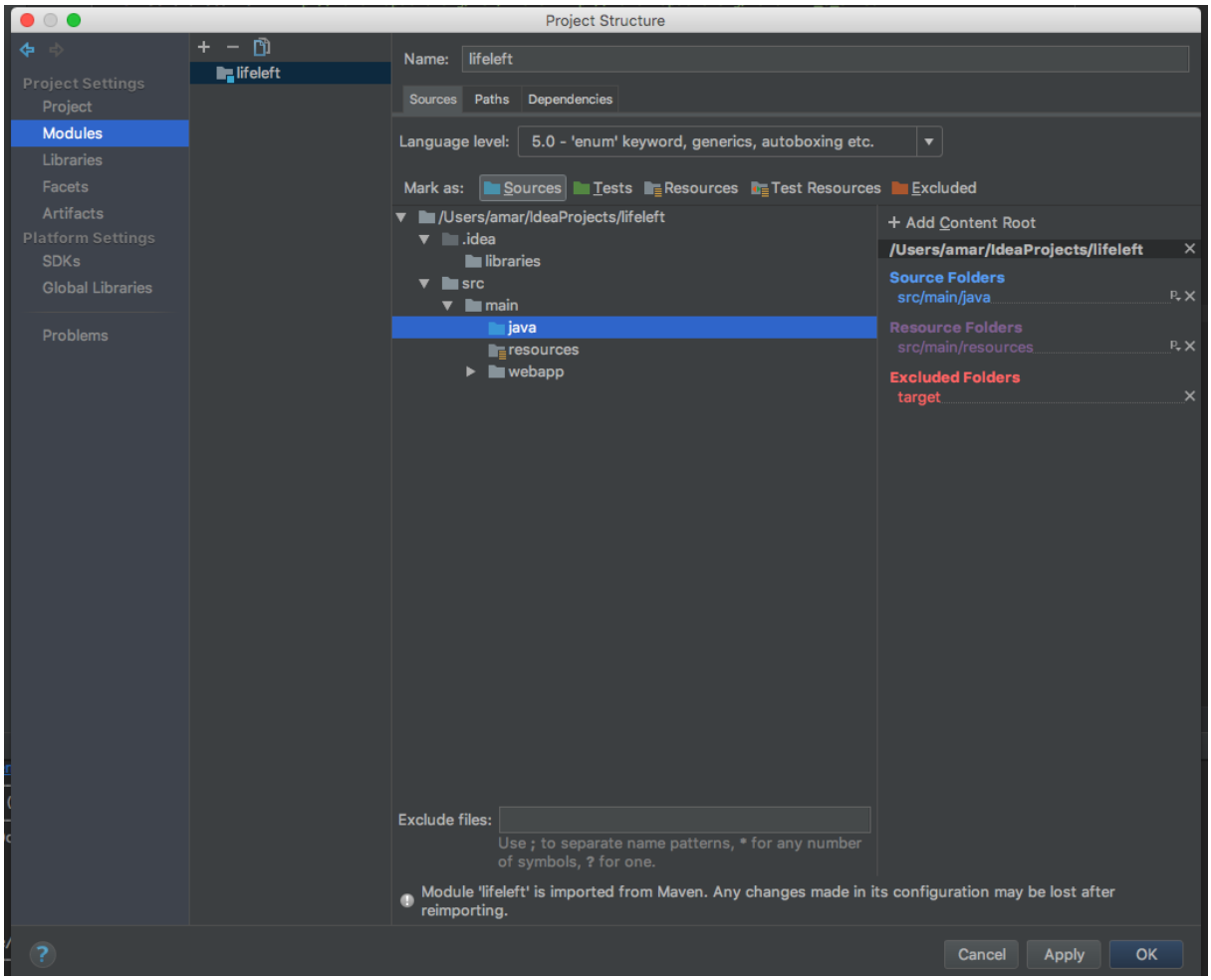


Figure 17

Créez un package et nommez le com.lifeleft : faites un clic droit sur *java* puis *New* > *Package* et entrez com.lifeleft. Vous obtenez ceci.

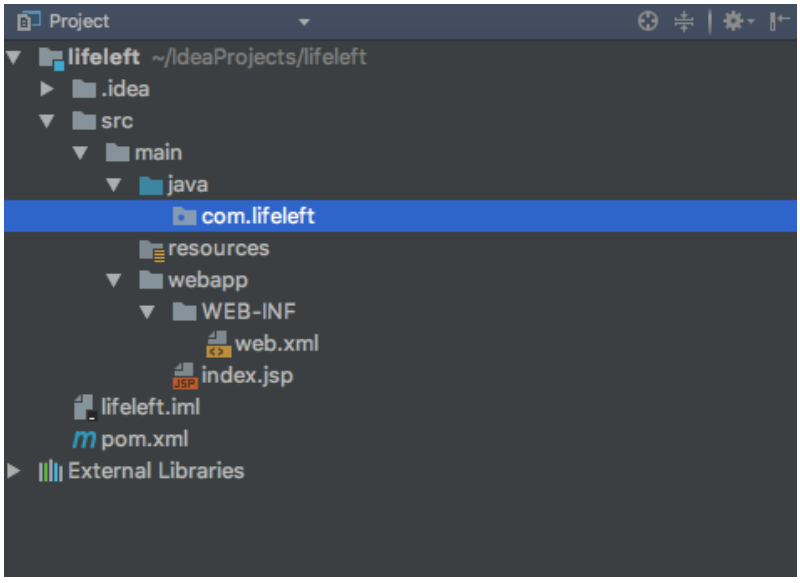


Figure 18

Supprimez le fichier web.xml que nous n'allons pas utiliser. (Il se trouve dans *webapp/WEB-INF/web.xml*.)

Très bien ! Vous avez maintenant tout ce qu'il faut pour annoncer de bonnes ou de mauvaises nouvelles sur la durée de vie de vos futurs utilisateurs. Développons le service !

Test : Evaluation compétences pratiques

Merci de suivre les étapes pratiques dans la partie précédente, pour l'installation des outils nécessaires et de :

1. Enregistrer une vidéo pour votre bureau durant votre travail sur l'installation de A jusqu'à Z.
2. M'envoyé les vidéos sur mon adresse émail (labidi8mohamed@gmail.com) en indiquant votre nom et prénom.